DEPARTMENT OF MATHEMATICS AND COMPUTING

V-M.Tech. (M&C)

Monsoon Semester 2022-2023

GPU Computing Lab MCC302

LAB-5

Makefile

NAME: **ROMEO SARKAR**
ADMISSION NO.: **20JE0814**
DATE: **07-08-2022**

Experiment 1.1: Use of Makefile with Main program, Distance Kernel, and Header Kernel.

Objectives: Use of Makefile

CUDA Sample Program:

```
#include "DistKernel.h"
#include <stdlib.h>
#define N 16
float scale (int i, int n)
{
    return ((float) (i)) / (n - 1);
}
int main ()
{
    const float ref = 0.5f;
    float *in = (float *) calloc (N, sizeof (float));
    float *out = (float *) calloc (N, sizeof (float));
    // compute scaled input values
    for (int i = 0; i < N; i++)
    {
        in[i] = scale (i, N);
    }
    // compute distances for the entire array
    distanceArray (out, in, ref, N);
    free (in);
    free (out);
    return 0;
}
```

distanceMain.cpp

```
#include "DistKernel.h"
#include <stdio.h>

#define TPB 16
__device__ float distance (float x1, float x2)
{
    return sqrt ((x2 - x1) * (x2 - x1));
}
__global__ void distanceKernel (float *d_out, float *d_in, float ref)
{
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    const float x = d_in[i];
    d_out[i] = distance (x, ref);
    printf ("i = %2d: distance from %f to %f is %f.\n", i, ref, x, d_out[i]);
    return;
}
void distanceArray (float *out, float *in, float ref, int len)
{
    // declare pointers to device arrays
    float *d_in = 0;
```

```
    float *d_out = 0;
    // allocate memory for device arrays
    cudaMalloc (&d_in, len * sizeof (float));
    cudaMalloc (&d_out, len * sizeof (float));
    // copy input data from host to device
    cudaMemcpy (d_in, in, len * sizeof (float), cudaMemcpyHostToDevice);
    // launch kernel to compute and store distance values
    distanceKernel <<<len / TPB, TPB>>> (d_out, d_in, ref);
    cudaDeviceSynchronize ();
    cudaMemcpy (out, d_out, len * sizeof (float), cudaMemcpyDeviceToHost);
    // free the memory allocated for device arrays
    cudaFree (d_in);
    cudaFree (d_out);
}
```

DistKernel.cu

```
#ifndef KERNEL_H
#define KERNEL_H
void distanceArray (float *out, float *in, float ref, int len);
#endif
```

DistKernel.h

```
NVCC = nvcc.exe

all: distanceMain.exe
distanceMain.exe: distanceMain.obj DistKernel.obj
    $(NVCC) $^ -o $@

distanceMain.obj: distanceMain.cpp
    $(NVCC) -c $^ -o $@

DistKernel.obj: DistKernel.cu
    $(NVCC) -c $^ -o $@
```

Makefile

Output:

```
i =  0: distance from 0.500000 to 0.000000 is 0.500000.
i =  1: distance from 0.500000 to 0.066667 is 0.433333.
i =  2: distance from 0.500000 to 0.133333 is 0.366667.
i =  3: distance from 0.500000 to 0.200000 is 0.300000.
i =  4: distance from 0.500000 to 0.266667 is 0.233333.
i =  5: distance from 0.500000 to 0.333333 is 0.166667.
i =  6: distance from 0.500000 to 0.400000 is 0.100000.
i =  7: distance from 0.500000 to 0.466667 is 0.033333.
i =  8: distance from 0.500000 to 0.533333 is 0.033333.
i =  9: distance from 0.500000 to 0.600000 is 0.100000.
i = 10: distance from 0.500000 to 0.666667 is 0.166667.
i = 11: distance from 0.500000 to 0.733333 is 0.233333.
i = 12: distance from 0.500000 to 0.800000 is 0.300000.
i = 13: distance from 0.500000 to 0.866667 is 0.366667.
i = 14: distance from 0.500000 to 0.933333 is 0.433333.
i = 15: distance from 0.500000 to 1.000000 is 0.500000.
```

Lab Exercise 1.1: Write a CUDA program to demonstrate the followings:

1) Write a header file for declaring functions (device and global).
2) Write a header file to transpose of Matrix A in GPU.
3) Then find the product of A and A^T using global functions.
4) Transfer result from device to host.
5) Print the result.

CODE:

```
#include <stdio.h>
#include "Matrix.cuh"
int main ()
{
    srand (time (NULL));
    Matrix M1 (3, 5), M2 (3, 5);
    M1.init (), M2.init ();
    Matrix Sum = M1 + M2;
    printf ("Matrix M1:\n");
    M1.display ();
    printf ("Matrix M2:\n");
    M2.display ();
    printf ("Matrix Sum:\n");
    Sum.display ();
    cudaDeviceSynchronize ();
    return 0;
}
```

Main.cu

```
#include <stdio.h>
#include <cuda_runtime.h>
#include "Matrix.cuh"
// macros:
#define precisionField 0
#define SHOW_FUNCTION_CALLS 1


Matrix :: Matrix () : rows (0), cols (0), device_pointer (NULL), host_pointer
(NULL)
{
    return;
}
Matrix :: Matrix (int r, int c) : Matrix ()
{
    rows = r;
    cols = c;
    alloc ();
    return;
}
Matrix :: Matrix (const Matrix &M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix (const Matrix &M)\033[m\n");
    #endif
    rows = M.rows;
    cols = M.cols;
    cudaMalloc (&device_pointer, rows * cols * sizeof (double));
    cudaMemcpy (device_pointer, M.device_pointer, rows * cols * sizeof (double),
cudaMemcpyDeviceToDevice);
    host_pointer = (double *) (malloc (rows * cols * sizeof (double)));
    memcpy (host_pointer, M.host_pointer, rows * cols * sizeof (double));
    return;
}
Matrix :: Matrix (Matrix &&M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix (Matrix &&M)\033[m\n");
    #endif
    rows = M.rows;
    cols = M.cols;
```

```
    device_pointer = M.device_pointer;
    host_pointer = M.host_pointer;
    M.rows = M.cols = 0;
    M.device_pointer = M.host_pointer = NULL;
    return;
}
Matrix Matrix :: operator = (Matrix &M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix operator = (Matrix &M)\033[m\n");
    #endif
    clear ();
    rows = M.rows;
    cols = M.cols;
    cudaMalloc (&device_pointer, rows * cols * sizeof (double));
    cudaMemcpy (device_pointer, M.device_pointer, rows * cols * sizeof (double),
cudaMemcpyDeviceToDevice);
    host_pointer = (double *) (malloc (rows * cols * sizeof (double)));
    memcpy (host_pointer, M.host_pointer, rows * cols * sizeof (double));
    return *this;
}
Matrix Matrix :: operator = (Matrix &&M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix operator = (Matrix &&M)\033[m\n");
    #endif
    rows = M.rows;
    cols = M.cols;
    device_pointer = M.device_pointer;
    host_pointer = M.host_pointer;
    M.rows = M.cols = 0;
    M.device_pointer = M.host_pointer = NULL;
    return *this;
}
Matrix :: ~Matrix ()
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90m~Matrix () : %p, %p\033[m\n", device_pointer, host_pointer);
    #endif
    if (NULL != device_pointer)
    {
        cudaFree (device_pointer);
    }
    if (NULL != host_pointer)
    {
        free (host_pointer);
    }
    rows = cols = 0;
    device_pointer = host_pointer = NULL;
    return;
}
void Matrix :: alloc ()
{
    cudaMalloc (&device_pointer, rows * cols * sizeof (double));
    host_pointer = (double *) (malloc (rows * cols * sizeof (double)));
    // printf ("hello");
    return;
}
void Matrix :: clear ()
{
    // printf ("%p, %p\n", device_pointer, host_pointer);
    if (NULL != device_pointer)
    {
        cudaFree (device_pointer);
    }
    if (NULL != host_pointer)
    {
        free (host_pointer);
    }
    rows = cols = 0;
```

```
    device_pointer = host_pointer = NULL;
    return;
}
void Matrix :: display ()
{
    if (NULL == host_pointer)
    {
        #if WARNINGS == 1
        printf ("\nIn function \'\e[33mprint_matrix_yu\e[m\':\n\e[35mwarning:\e[m
\'m\' is (null)\n");
        #endif
        return;
    }
    #define BUFFER_SIZE 128
    // double (*mat)[cols] = (double (*)[cols]) (host_pointer);
    int *max_width_arr = (int *) (malloc (cols * sizeof (int)));
    char **mat_of_strs = (char **) malloc (rows * cols * sizeof (char *));
    // char *(*matrix_of_strings)[c] = mat_of_strs;
    char *str;
    int width;
    for (size_t i = 0; i < cols; i++)
    {
        max_width_arr[i] = 1;
        for (size_t j = 0; j < rows; j++)
        {
            str = (char *) malloc (BUFFER_SIZE * sizeof (char));
            width = snprintf (str, BUFFER_SIZE, "%.*lf", precisionField,
host_pointer[j * cols + i]);
            str = (char *) realloc (str, ((size_t) (width + 1)) * sizeof (char));
            mat_of_strs[j * cols + i] = str;
            if (max_width_arr[i] < width)
                max_width_arr[i] = width;
        }
    }
    for (size_t i = 0; i < rows; i++)
    {
        printf ("\033[1;32m\xb3\033[m");
        for (size_t j = 0; j < cols; j++)
        {
            width = strlen (mat_of_strs[i * cols + j]);
            for (int x = 0; x < max_width_arr[j] - width; x++)
                printf (" ");
            printf ("%s", mat_of_strs[i * cols + j]);
            if (j != (cols - 1))
                printf (" ");
        }
        printf ("\033[1;32m\xb3\033[m");
        // newline:
        printf ("\n");
    }
    for (size_t i = 0; i < rows; i++)
        for (size_t j = 0; j < cols; j++)
            free (mat_of_strs[i * cols + j]);
    free (mat_of_strs);
    free (max_width_arr);
    return;
}
void Matrix :: init ()
{
    ::init (host_pointer, rows, cols);
    // cudaDeviceSynchronize ();
    // printf ("\033[31mhere\033[m");
    H2D ();
    // printf ("here");
    return;
}
void Matrix :: H2D ()
{
    cudaMemcpy (device_pointer, host_pointer, cols * rows * sizeof (double),
cudaMemcpyHostToDevice);
```

```
    return;
}
void Matrix :: D2H ()
{
    cudaMemcpy (host_pointer, device_pointer, cols * rows * sizeof (double),
cudaMemcpyDeviceToHost);
    return;
}
Matrix Matrix :: operator + (const Matrix &M)
{
    if (rows != M.rows && cols != M.cols)
    {
        printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);
        return Matrix ();
    }
    Matrix p (rows, M.cols);
    dim3 block (1, 1, 1);
    dim3 grid (rows, M.cols, 1);
    add_GPU <<< block, grid>>> (device_pointer, M.device_pointer,
p.device_pointer, rows, cols);
    cudaDeviceSynchronize ();
    p.D2H ();
    // p.display ();
    return p;
}
Matrix Matrix :: operator - (const Matrix &M)
{
    if (rows != M.rows && cols != M.cols)
    {
        printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);
        return Matrix ();
    }
    Matrix p (rows, M.cols);
    dim3 block (1, 1, 1);
    dim3 grid (rows, M.cols, 1);
    sub_GPU <<< block, grid>>> (device_pointer, M.device_pointer,
p.device_pointer, rows, cols);
    cudaDeviceSynchronize ();
    p.D2H ();
    // p.display ();
    return p;
}
Matrix Matrix :: operator * (const Matrix &M)
{
    if (cols != M.rows)
    {
        printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);
        return Matrix ();
    }
    Matrix p (rows, M.cols);
    dim3 block (1, 1, 1);
    dim3 grid (rows, M.cols, 1);
    mul_GPU <<< block, grid>>> (device_pointer, M.device_pointer,
p.device_pointer, rows, cols, M.cols);
    cudaDeviceSynchronize ();
    p.D2H ();
    // p.display ();
    return p;
}
Matrix Matrix :: operator ~ ()
{
    Matrix t (cols, rows);
    dim3 block (1, 1, 1);
    dim3 grid (rows, cols, 1);
    trp_GPU <<<grid, block>>> (device_pointer, t.device_pointer, rows, cols);
    cudaDeviceSynchronize ();
    t.D2H ();
    return t;
}
```

```
__global__ void init_GPU (double *p, int rows, int cols)
{
    int r = threadIdx.x + blockIdx.x * blockDim.x; // x = rows
    int c = threadIdx.y + blockIdx.y * blockDim.y; // y = cols
    // printf ("%d;%d;%d;%d\n", r, c, M.rows, M.cols);
    if (r < rows && c < cols)
    {
        // printf ("<%d>", r * M.cols + c);
        p[r * cols + c] = ((double) (r * cols + c));
        // printf ("%lf ", M.device_pointer[r * M.cols + c]);
    }
    return;
}
void init (double *p, int rows, int cols)
{
    for (int i = 0; i < rows * cols; i++)
    {
        p[i] = rand () % 21 - 10;
    }
    return;
}
__device__ double add_GPU_dev (double m1, double m2)
{
    return m1 + m2;
}
__global__ void add_GPU (double *m1, double *m2, double *a, int rows, int cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
    {
        a[Row * cols + Col] = add_GPU_dev (m1[Row * cols + Col], m2[Row * cols +
Col]);
    }
    return;
}
__global__ void sub_GPU (double *m1, double *m2, double *a, int rows, int cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
    {
        a[Row * cols + Col] = m1[Row * cols + Col] - m2[Row * cols + Col];
    }
    return;
}
__global__ void mul_GPU (double *m1, double *m2, double *p, int rows, int x, int
cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
    {
        // printf ("{%d,%d}", Row, Col);
        double a = 0;
        for (int k = 0; k < x; k++)
        {
            // printf ("(%.0f,%.0f)", m1[Row * cols + k], m2[k * rows + Col]);
            a += m1[Row * x + k] * m2[k * cols + Col];
        }
        p[Row * cols + Col] = a;
        // printf ("=<%f>\n", a);
    }
    return;
}
__global__ void trp_GPU (double *m1, double *m2, int rows, int cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
```

```
    {
        m2[Col * rows + Row] = m1[Row * cols + Col];
    }
    return;
}
```

Matrix.cu

```
__global__ void init_GPU (double *p, int rows, int cols);
__global__ void mul_GPU (double *m1, double *m2, double *p, int rows, int x, int
cols);
__global__ void trp_GPU (double *m1, double *m2, int rows, int cols);
__global__ void add_GPU (double *m1, double *m2, double *a, int rows, int cols);
__global__ void sub_GPU (double *m1, double *m2, double *a, int rows, int cols);
void init (double *p, int rows, int cols);
struct Matrix
{
    int rows, cols;
    double *device_pointer, *host_pointer;
    int flag = 0;
    Matrix ();
    Matrix (int r, int c);
    Matrix (const Matrix &M);
    Matrix (Matrix &&M);
    Matrix operator = (Matrix &M);
    Matrix operator = (Matrix &&M);
    ~Matrix ();
    void alloc ();
    void clear ();
    void display ();
    void init ();
    void H2D ();
    void D2H ();
    Matrix operator + (const Matrix &M);
    Matrix operator - (const Matrix &M);
    Matrix operator * (const Matrix &M);
    Matrix operator ~ ();
};
```

Matrix.cuh

```
CC = nvcc
FLAGS = -dc -c

# Targets = Main.cu Matrix.cu
ALL: Lib\Main.obj Lib\Matrix.obj
    $(CC) Lib\Main.obj Lib\Matrix.obj -o Main
    .\Main.exe

Lib\Main.obj: Main.cu
    $(CC) $(FLAGS) Main.cu -o "Lib/Main"
Lib\Matrix.obj: Matrix.cu
    $(CC) $(FLAGS) Matrix.cu -o "Lib/Matrix"

CLEAN:
    del "Lib\*.obj"
    del "Main.exe"
    del "Main.lib"
    del "Main.exp"
```

Makefile

Output:

```
.\Main.exe
Matrix M1:
|10 2 0   5 -5|
|-3 2 1   1 -1|
|-4 4 4 -10 -5|
Matrix M2:
|-5  6  -8 -4  2|
| 5 -9 -10  4 -6|
|-3 -3  -9 -8  3|
Matrix Sum:
| 5  8 -8   1 -3|
| 2 -7 -9   5 -7|
|-7  1 -5 -18 -2|
```

Lab Exercise 1.2: Write a CUDA program to demonstrate:

1. Write a header file for declaring functions.
2. Write device functions to transpose of Matrix A in GPU.
3. Then find the product of A and A^T using global functions.
4. Transfer results from device to host.
5. Print the result.

CODE:

```
#include <stdio.h>
#include "matrix.cuh"
int main ()
{
    srand (time (NULL));
    Matrix A (4, 3);
    A.init ();
    Matrix AT = ~A;
    printf ("Matrix A:\n");
    A.display ();
    printf ("Matrix AT:\n");
    AT.display ();
    Matrix P = A * AT;
    printf ("Matrix P:\n");
    P.display ();
    cudaDeviceReset ();
    return 0;
}
```

Main.cu

```
#include <stdio.h>
#include <cuda_runtime.h>
#include "matrix.cuh"
// macros:
#define precisionField 0
#define SHOW_FUNCTION_CALLS 1

Matrix :: Matrix () : rows (0), cols (0), device_pointer (NULL), host_pointer
(NULL)
{
    return;
}
Matrix :: Matrix (int r, int c) : Matrix ()
{
    rows = r;
    cols = c;
    alloc ();
    return;
}
Matrix :: Matrix (const Matrix &M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix (const Matrix &M)\033[m\n");
    #endif
    rows = M.rows;
    cols = M.cols;
    cudaMalloc (&device_pointer, rows * cols * sizeof (double));
    cudaMemcpy (device_pointer, M.device_pointer, rows * cols * sizeof (double),
cudaMemcpyDeviceToDevice);
    host_pointer = (double *) (malloc (rows * cols * sizeof (double)));
    memcpy (host_pointer, M.host_pointer, rows * cols * sizeof (double));
    return;
}
Matrix :: Matrix (Matrix &&M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix (Matrix &&M)\033[m\n");
    #endif
    rows = M.rows;
    cols = M.cols;
    device_pointer = M.device_pointer;
    host_pointer = M.host_pointer;
```

```
    M.rows = M.cols = 0;
    M.device_pointer = M.host_pointer = NULL;
    return;
}
Matrix Matrix :: operator = (Matrix &M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix operator = (Matrix &M)\033[m\n");
    #endif
    clear ();
    rows = M.rows;
    cols = M.cols;
    cudaMalloc (&device_pointer, rows * cols * sizeof (double));
    cudaMemcpy (device_pointer, M.device_pointer, rows * cols * sizeof (double),
cudaMemcpyDeviceToDevice);
    host_pointer = (double *) (malloc (rows * cols * sizeof (double)));
    memcpy (host_pointer, M.host_pointer, rows * cols * sizeof (double));
    return *this;
}
Matrix Matrix :: operator = (Matrix &&M)
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90mMatrix operator = (Matrix &&M)\033[m\n");
    #endif
    rows = M.rows;
    cols = M.cols;
    device_pointer = M.device_pointer;
    host_pointer = M.host_pointer;
    M.rows = M.cols = 0;
    M.device_pointer = M.host_pointer = NULL;
    return *this;
}
Matrix :: ~Matrix ()
{
    #if SHOW_FUNCTION_CALLS == 1
    printf ("\033[90m~Matrix () : %p, %p\033[m\n", device_pointer, host_pointer);
    #endif
    if (NULL != device_pointer)
    {
        cudaFree (device_pointer);
    }
    if (NULL != host_pointer)
    {
        free (host_pointer);
    }
    rows = cols = 0;
    device_pointer = host_pointer = NULL;
    return;
}
void Matrix :: alloc ()
{
    cudaMalloc (&device_pointer, rows * cols * sizeof (double));
    host_pointer = (double *) (malloc (rows * cols * sizeof (double)));
    // printf ("hello");
    return;
}
void Matrix :: clear ()
{
    // printf ("%p, %p\n", device_pointer, host_pointer);
    if (NULL != device_pointer)
    {
        cudaFree (device_pointer);
    }
    if (NULL != host_pointer)
    {
        free (host_pointer);
    }
    rows = cols = 0;
    device_pointer = host_pointer = NULL;
    return;
```

```
}
void Matrix :: display ()
{
    if (NULL == host_pointer)
    {
        #if WARNINGS == 1
        printf ("\nIn function \'\e[33mprint_matrix_yu\e[m\':\n\e[35mwarning:\e[m
\'m\' is (null)\n");
        #endif
        return;
    }
    #define BUFFER_SIZE 128
    // double (*mat)[cols] = (double (*)[cols]) (host_pointer);
    int *max_width_arr = (int *) (malloc (cols * sizeof (int)));
    char **mat_of_strs = (char **) malloc (rows * cols * sizeof (char *));
    // char *(*matrix_of_strings)[c] = mat_of_strs;
    char *str;
    int width;
    for (size_t i = 0; i < cols; i++)
    {
        max_width_arr[i] = 1;
        for (size_t j = 0; j < rows; j++)
        {
            str = (char *) malloc (BUFFER_SIZE * sizeof (char));
            width = snprintf (str, BUFFER_SIZE, "%.*lf", precisionField,
host_pointer[j * cols + i]);
            str = (char *) realloc (str, ((size_t) (width + 1)) * sizeof (char));
            mat_of_strs[j * cols + i] = str;
            if (max_width_arr[i] < width)
                max_width_arr[i] = width;
        }
    }
    for (size_t i = 0; i < rows; i++)
    {
        printf ("\033[1;32m\xb3\033[m");
        for (size_t j = 0; j < cols; j++)
        {
            width = strlen (mat_of_strs[i * cols + j]);
            for (int x = 0; x < max_width_arr[j] - width; x++)
                printf (" ");
            printf ("%s", mat_of_strs[i * cols + j]);
            if (j != (cols - 1))
                printf (" ");
        }
        printf ("\033[1;32m\xb3\033[m");
        // newline:
        printf ("\n");
    }
    for (size_t i = 0; i < rows; i++)
        for (size_t j = 0; j < cols; j++)
            free (mat_of_strs[i * cols + j]);
    free (mat_of_strs);
    free (max_width_arr);
    return;
}
void Matrix :: init ()
{
    ::init (host_pointer, rows, cols);
    // cudaDeviceSynchronize ();
    // printf ("\033[31mhere\033[m");
    H2D ();
    // printf ("here");
    return;
}
void Matrix :: H2D ()
{
    cudaMemcpy (device_pointer, host_pointer, cols * rows * sizeof (double),
cudaMemcpyHostToDevice);
    return;
}
```

```
void Matrix :: D2H ()
{
    cudaMemcpy (host_pointer, device_pointer, cols * rows * sizeof (double),
cudaMemcpyDeviceToHost);
    return;
}
Matrix Matrix :: operator + (const Matrix &M)
{
    if (rows != M.rows && cols != M.cols)
    {
        printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);
        return Matrix ();
    }
    Matrix p (rows, M.cols);
    dim3 block (1, 1, 1);
    dim3 grid (rows, M.cols, 1);
    add_GPU <<< block, grid>>> (device_pointer, M.device_pointer,
p.device_pointer, rows, cols);
    cudaDeviceSynchronize ();
    p.D2H ();
    // p.display ();
    return p;
}
Matrix Matrix :: operator - (const Matrix &M)
{
    if (rows != M.rows && cols != M.cols)
    {
        printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);
        return Matrix ();
    }
    Matrix p (rows, M.cols);
    dim3 block (1, 1, 1);
    dim3 grid (rows, M.cols, 1);
    sub_GPU <<< block, grid>>> (device_pointer, M.device_pointer,
p.device_pointer, rows, cols);
    cudaDeviceSynchronize ();
    p.D2H ();
    // p.display ();
    return p;
}
Matrix Matrix :: operator * (const Matrix &M)
{
    if (cols != M.rows)
    {
        printf ("Matrix1 (%dX%d); Matrix2 (%dX%d)\n", rows, cols, M.rows, M.cols);
        return Matrix ();
    }
    Matrix p (rows, M.cols);
    dim3 block (1, 1, 1);
    dim3 grid (rows, M.cols, 1);
    mul_GPU <<< block, grid>>> (device_pointer, M.device_pointer,
p.device_pointer, rows, cols, M.cols);
    cudaDeviceSynchronize ();
    p.D2H ();
    // p.display ();
    return p;
}
Matrix Matrix :: operator ~ ()
{
    Matrix t (cols, rows);
    dim3 block (1, 1, 1);
    dim3 grid (rows, cols, 1);
    trp_GPU <<<grid, block>>> (device_pointer, t.device_pointer, rows, cols);
    cudaDeviceSynchronize ();
    t.D2H ();
    return t;
}

__global__ void init_GPU (double *p, int rows, int cols)
{
```

```
    int r = threadIdx.x + blockIdx.x * blockDim.x; // x = rows
    int c = threadIdx.y + blockIdx.y * blockDim.y; // y = cols
    // printf ("%d;%d;%d;%d\n", r, c, M.rows, M.cols);
    if (r < rows && c < cols)
    {
        // printf ("<%d>", r * M.cols + c);
        p[r * cols + c] = ((double) (r * cols + c));
        // printf ("%lf ", M.device_pointer[r * M.cols + c]);
    }
    return;
}
void init (double *p, int rows, int cols)
{
    for (int i = 0; i < rows * cols; i++)
    {
        p[i] = rand () % 21 - 10;
    }
    return;
}
__device__ double add_GPU_dev (double m1, double m2)
{
    return m1 + m2;
}
__global__ void add_GPU (double *m1, double *m2, double *a, int rows, int cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
    {
        a[Row * cols + Col] = add_GPU_dev (m1[Row * cols + Col], m2[Row * cols +
Col]);
    }
    return;
}
__global__ void sub_GPU (double *m1, double *m2, double *a, int rows, int cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
    {
        a[Row * cols + Col] = m1[Row * cols + Col] - m2[Row * cols + Col];
    }
    return;
}
__global__ void mul_GPU (double *m1, double *m2, double *p, int rows, int x, int
cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
    {
        // printf ("{%d,%d}", Row, Col);
        double a = 0;
        for (int k = 0; k < x; k++)
        {
            // printf ("(%.0f,%.0f)", m1[Row * cols + k], m2[k * rows + Col]);
            a += m1[Row * x + k] * m2[k * cols + Col];
        }
        p[Row * cols + Col] = a;
        // printf ("=<%f>\n", a);
    }
    return;
}
__device__ double trp_GPU_dev (double *m, int cols, int Row, int Col)
{
    return m[Row * cols + Col];
}
__global__ void trp_GPU (double *m1, double *m2, int rows, int cols)
{
    int Row = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    int Col = blockIdx.y * blockDim.y + threadIdx.y;
    if (Row < rows && Col < cols)
    {
        m2[Col * rows + Row] = trp_GPU_dev (m1, cols, Row, Col);
    }
    return;
}
```

Matrix.cu

```
__global__ void init_GPU (double *p, int rows, int cols);
__global__ void mul_GPU (double *m1, double *m2, double *p, int rows, int x, int
cols);
__global__ void trp_GPU (double *m1, double *m2, int rows, int cols);
__global__ void add_GPU (double *m1, double *m2, double *a, int rows, int cols);
__global__ void sub_GPU (double *m1, double *m2, double *a, int rows, int cols);
void init (double *p, int rows, int cols);
struct Matrix
{
    int rows, cols;
    double *device_pointer, *host_pointer;
    int flag = 0;
    Matrix ();
    Matrix (int r, int c);
    Matrix (const Matrix &M);
    Matrix (Matrix &&M);
    Matrix operator = (Matrix &M);
    Matrix operator = (Matrix &&M);
    ~Matrix ();
    void alloc ();
    void clear ();
    void display ();
    void init ();
    void H2D ();
    void D2H ();
    Matrix operator + (const Matrix &M);
    Matrix operator - (const Matrix &M);
    Matrix operator * (const Matrix &M);
    Matrix operator ~ ();
};
```

Matrix.cuh

```
CC = nvcc
FLAGS = -dc -c

# Targets = Main.cu Matrix.cu
ALL: Lib\Main.obj Lib\Matrix.obj
    $(CC) Lib\Main.obj Lib\Matrix.obj -o Main
    .\Main.exe

Lib\Main.obj: Main.cu
    $(CC) $(FLAGS) Main.cu -o "Lib/Main"
Lib\Matrix.obj: Matrix.cu
    $(CC) $(FLAGS) Matrix.cu -o "Lib/Matrix"

CLEAN:
    del "Lib\*.obj"
    del "Main.exe"
    del "Main.lib"
    del "Main.exp"
```

Makefile

Outputs:

```
.\Main.exe
Matrix A:
| 7  2  0|
| 6  2  2|
|-3  9 -3|
| 5 -7  5|
Matrix AT:
|7 6 -3  5|
|2 2  9 -7|
|0 2 -3  5|
Matrix P:
|53 46  -3  21|
|46 44  -6  26|
|-3 -6  99 -93|
|21 26 -93  99|
```